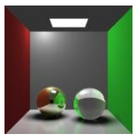


2016/2017, 4th quarter

INFOGR: Graphics

Practical 2: Ray Tracing

Author: Jacco Bikker

The assignment:

The purpose of this assignment is to create a small Whitted-style ray tracer. The renderer should be able to render a scene consisting of reflecting or diffuse spheres and planes, illuminated by point lights. For a full list of required functionality, see section “The Full Thing”.

As with the first assignment, the following rules for submission apply:

- Your code has to compile and run on the machines in the rooms allocated to the working colleges, so if you work on other computers make sure to do a quick check there before you submit it. If this requirement isn't met, your work cannot be graded and your grade will default to 0.
- Please **clean** your solution before submitting (i.e. remove all the compiled files and intermediate output). This can easily be achieved by running clean.bat (included with the template). After this you can zip the solution directories and send them over. If your zip-file is multiple mega-bytes in size something went wrong (not cleaned properly).
- When grading, we want to get the impression that you really understand what is happening in your code, so your source files should also contain comments to explain what you think is happening.
- Finally, we also want to see a consistent and readable coding style. Use indentation to indicate structure in the code for example. Don't worry about this too much, if it is readable and consistent throughout the whole project, you should be fine.

Grading:

If you implement the minimum requirements, and stick to the above rules, you get a 6. Implement additional features to obtain additional points (up to a 10).

Additional grading details:

- From the base grade of 6, we deduct points for a missing readme, a solution that was not cleaned, a solution that does not compile, or a solution that crashes (1 point for each problem).
- One point will be deducted if you worked alone on the assignment.
- Up to 1 point will be deducted for an inconsistent coding style.
- Up to 1 point will be deducted for code that is not properly commented.

Deliverables:

A ZIP-file containing:

1. **The contents of your (cleaned) solution directory**
2. **The read-me (in the .txt file format)**

The contents of the solution directory should contain:

(a) Your **solution file** (.sln)

(b) All your **source code**

(c) All your **project** and **content files**

The readme file should contain:

(a) The names and student IDs of your team members.

[2-3 students; penalties for submitting with less or more team members apply]

(b) A statement about what bonus assignments you have implemented (if any) and related information that is needed to grade them, including detailed information on your implementation.

[We will not make any wild guesses about what you might have implemented nor will we spend long times searching for special features in your code. If we can't find and understand them easily, they will not be graded, so make sure your description and/or comments are clear.]

(c) A list of materials you used to implement the ray tracer. If you borrowed code or ideas, make sure you provide a full and accurate overview of this. Considering the large number of ray tracers available on the internet, we will carefully check for original work.

Put the solution directories and the read-me file (in the .txt file format) directly in the **root** of the zip file. Note that any violation to these rules can have consequences for your grade. Also notice that the readme file should be well readable. It is part of the program that you are producing, so the rules about "consistent and clear coding style" apply to it as well.

Mode of submission:

- Upload your zip file before the deadline via the SUBMIT system at http://www.cs.uu.nl/docs/submit/
- Make sure to upload it to the correct entry, i.e. **not** the ones for late delivery if you are submitting on time (otherwise, grade deductions will still apply).
- Note that we only grade the latest submitted version of your assignment, so if you upload to the late delivery entries your earlier submission will be discarded.

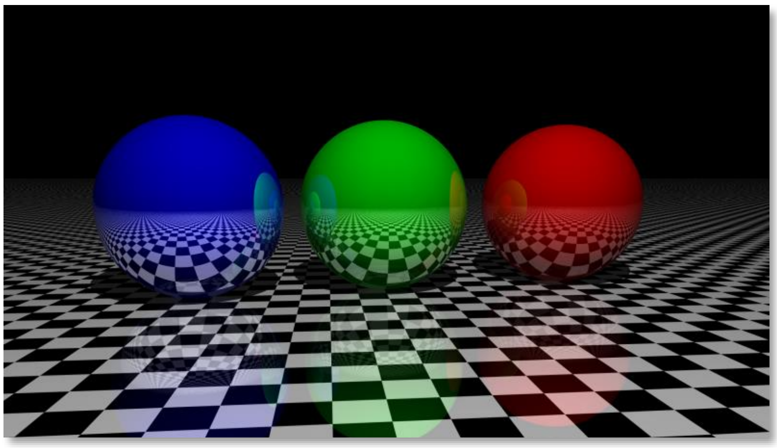
Deadline:

Tuesday, May 30, 2017, 23:59h

If you miss this deadline, there will be a second entry in the submit system to upload your solutions. It is open 12h longer, i.e. till **Wednesday, May 31, 2017, 12:00 (noon)**. Uploading to this entry will result in a deduction of 0.5 in your grading (attention: the deduction still applies if you upload to this entry earlier).

If you miss the second deadline as well, there will be a third entry in the submit system to upload your solutions. It is open 24h longer, i.e. till **Wednesday, May 31, 2017, 23:59**. Uploading to this entry will result in a deduction of 1.0 in your grading (attention: the deduction still applies if you upload to this entry earlier).

If you the miss third deadline as well, your assignment will be graded with a 0.



High-level Outline

For this assignment you will implement a Whitted-style ray tracer. This is a recursive rendering algorithm for determining light transport between one or more light sources and a camera, via scene surfaces, by tracing rays backwards into the scene, starting at the camera.

A Whitted-style ray tracer requires a number of basic ingredients:

- A camera, representing the position and direction of the observer in the virtual world;
- A screen plane floating in front of the camera, which will be used to fire rays at;
- A number of primitives that will be intersected by the camera rays;
- A number of light sources that provide the energy that will be transported back to the camera;
- A renderer that serves as the access point from the main application. The renderer 'owns' the camera and scene, generates the rays, intersects them with the scene, determines the nearest intersection, and plots pixels based on calculated light transport.

Optionally, we can define materials for the primitives. A material stores information about color or texture, reflectivity, refractivity and absorption.

The remainder of this document describes the implementation of such a ray tracer. You are free to ignore these steps and go straight to the required feature list. Note that the debug view is a mandatory feature.

Architecture

To start this project, create classes for the fundamental elements of the ray tracer:

Camera, with data members position and direction. The camera also stores the screen plane, specified by its four corners, which are updated whenever camera position and/or direction is modified. Hardcoded coordinates and directions allow for an easy start. Use e.g. (0,0,0) as the camera origin, and (0,0,1) as the direction; this way the screen corners can also be hardcoded for the time being. Once the basic setup works, you can make this more flexible.

Primitive, which encapsulates the ray/primitive intersection functionality. Two classes should be derived from the base primitive: Sphere and Plane. A sphere is defined by a position and a radius; a plane is defined by a normal and a distance to the origin. Initially (until you implement materials) it may also be useful to add a color to the primitive class.

Important: colors should be stored as floating point vectors. We will convert the final transported light quantities to integer color as a final step; keeping everything in floats is accurate, more natural, and easier.

Light, which stores the location and intensity of a light source. For a Whitted-style ray tracer, this will be a point light. Intensity should be stored using float values for red, green and blue.

Scene, which stores a list of primitives and light sources. It implements a scene-level Intersect method, which loops over the primitives and returns the closest intersection.

Intersection, which stores the result of an intersection. Apart from the intersection distance, you will at least want to store the nearest primitive, but perhaps also the normal at the intersection point.

Raytracer, which owns the scene, camera and the display surface. The Raytracer implements a method Render, which uses the camera to loop over the pixels of the screen plane and to generate a ray for each pixel, which is then used to find the nearest intersection. The result is then visualized by plotting a pixel. For one line of pixels (typically line 256 for a 512x512 window), it generates debug output by visualizing every N^{th} ray (where N is e.g. 10).

Application, which calls the Render method of the Raytracer. The application is responsible for handling keyboard and/or mouse input.

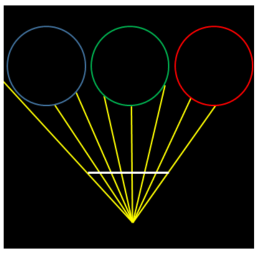
First Steps - Details

With the basic structure of the application in place, it is time to implement the functionality. It helps to get to something that produces sensible output as quickly as possible.

1. Prepare the scene.

A good scene to start with is a floor plane with three spheres on it. Keep everything within a 10x10x10 cube, and position the spheres so that the default camera can easily 'see' them. Make sure their centers are at y=0, which is convenient for the debug view.

2. Prepare the debug output.



Draw the scene to the debug output. Use a dot (or 2x2 dots) to visualize the position of the camera. Use a line to visualize the screen plane. Draw ~100 line segments to approximate the spheres. Use the coordinate system translation from the tutorial to get a view where camera and spheres fit in the 512x512 debug window. Skip the plane in the visualization.

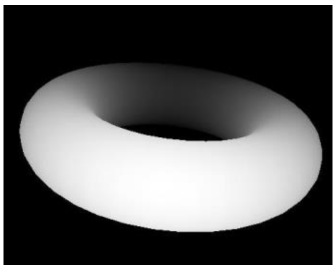
3. Generate primary rays.

Use the loop in the Raytracer.Render method to produce the primary rays. Note that you can use a single ray here: it can be reused once the pixel color has been found. In the debug window, draw a red line for the normalized ray direction. Verify that the generated rays form an arc with radius 1. Verify that no rays miss the screen plane.

4. Intersect the scene.

Use the primary rays to intersect the scene. Now the full rays can be displayed in the debug output. If you visualize rays for y=0 (i.e., line 256 of the 3D view), these rays should exactly end at the sphere boundaries.

5. Visualize the intersections.



Once you have an intersection for a primary ray, you can use its data to make up a pixel color. Plot a black pixel if the ray didn't hit anything. Otherwise, take the intersection distance, and scale it to a suitable range (i.e., the scaled distances should be in the range 0..1). Now you can use this value to plot a greyscale value. This should yield a 'depth map' of your scene.

From here on you can slowly implement the remaining features, but you will never have to work without visual feedback. E.g., normals can be visualized as little lines pointing away from intersection points, and shadow rays can be colored based on whether they hit an obstruction or not. Use the debug view extensively to verify your code.

The Full Thing

To pass this assignment, implement the following features:

Camera:

- Your camera must support arbitrary field of view. It must be possible to set FOV from the application, using an angle in degrees.
- The camera must support arbitrary positions and orientations. It must be possible to aim the camera based on an arbitrary 3D position and target.

Primitives:

- Your ray tracer must at least support planes and spheres. The scene definition must be flexible, i.e. your ray tracer must be able to handle arbitrary sets of planes and spheres.

Lights:

- Your ray tracer must be able to correctly handle an arbitrary number of point lights and their shadows.

Materials:

- Your ray tracer must support at least diffuse materials and colored specular materials (mirrors). The mirrors must be recursive, with an adjustable cap on recursion depth. There must at least be texturing for a single plane (e.g., the floor plane) to make correct reflections visible (and verifiable).

Application:

- The application must support keyboard and/or mouse handling to control the camera.

Debug output:

- The debug output must be implemented. It must at least show the primitives (except for infinite planes), primary rays, shadow rays and secondary rays.

Note that there is no performance requirement for this application.

A Bit Extra

Meeting the minimum requirements earns you a 6 (assuming practical details are all in order). An additional four points can be earned by implementing optional features. An incomplete list of options, with an indication of the difficulty level:

- [EASY] Add triangle support (1 pt) with optional normal interpolation (1 pt)
- [EASY] Add spotlights (1 pt)
- [EASY] Add stochastic sampling of glossy reflections (1 pt)
- [EASY] Add anti-aliasing (1 pt)
- [MEDIUM] Add textures to all primitives (1 pt) with optional normal maps (1 pt)
- [MEDIUM] Add a textured skydome (1 pt), make it HDR for an additional 1 pt
- [MEDIUM] Add refraction (1 pt) and absorption (1 pt)
- [MEDIUM] Add stochastic sampling of area lights (1 pt)
- [HARD] Add an acceleration structure (2 pts)
- [HARD] Implement the ray tracer on a GPU (2 pts)

Important: many of these features require that you investigate these yourself, i.e. they are not necessarily covered in the lectures. You may of course discuss these on Slack to get some help. Also note that performance-related functionality is generally not rewarded (one exception: see below).

On top of these, you may compete for two special rewards:

1. Fastest ray tracer

Use every trick from the book to make your ray tracer run as quickly as possible. GPGPU, multithreading, SIMD; everything is on the table. For fairness, we need to specify the scene: use a 5x5x5 room with three diffuse spheres of radius 1 in it, and position the camera so that the three spheres are all completely visible. Performance will be measured on my 24-core desktop. Win the award for **one bonus point (up to a 10)**. And glory. Lots of glory.

2. Smallest ray tracer

Reduce the size of your ray tracer to as few characters as possible. The resulting ray tracer must meet the minimum requirements of this assignment, except anything involving interactivity and the debug output. You may implement the tiny tracer in a programming language of your choice. Win the award for **one bonus point (up to a 10)** and pride.

2a: reward 2 interpreted as 'smallest source file'.
2b: reward 2 interpreted as 'smallest executable'.

And Finally…

Don't forget to have fun; make something beautiful!

May the Light be with you,

- *Jacco.*